Clone Detection in Repositories of Business Process Models

Reina Uba[1], Marlon Dumas[1], Luciano García-Bañuelos[1], and Marcello La Rosa[2]

[1] University of Tartu, Estonia
{reinak, marlon.dumas, luciano.garcia}@ut.ee
[2] Queensland University of Technology, Australia
m.larosa@qut.edu.au

Abstract. As organizations reach to higher levels of business process management maturity, they often find themselves maintaining repositories of hundreds or even thousands of process models, representing valuable knowledge about their operations. Over time, process model repositories tend to accumulate duplicate fragments (also called *clones*) as new process models are created or extended by copying and merging fragments from other models. This calls for methods to detect clones in process models, so that these clones can be refactored as separate subprocesses in order to improve maintainability. This paper presents an indexing structure to support the fast detection of clones in large process model repositories. The proposed index is based on a novel combination of a method for process model decomposition (specifically the Refined Process Structure Tree), with established graph canonization and string matching techniques. Experiments show that the algorithm scales to repositories with hundreds of models. The experimental results also show that a significant number of non-trivial clones can be found in process model repositories taken from industrial practice.

1 Introduction

It is nowadays common for organizations engaged in long-term business process management programs to deal with collections of hundreds or even thousands of process models, with sizes ranging from dozens to hundreds of elements per model [15, 14]. While highly valuable, such collections of process models raise a significant maintenance problem [14]. This problem is amplified by the fact that process models in large organizations are maintained and used by stakeholders with varying skills, responsibilities and goals, sometimes distributed across independent organizational units.

One problem that arises as repositories grow is that of managing overlaps across models. In particular, process model repositories tend to accumulate duplicate fragments over time, as new process models are created by copying and merging fragments from other models. Experiments conducted during this study show that a well-known process model repository contains several hundred non-trivial clones. This situation is akin to that observed in source code repositories, where significant amounts of duplicate code fragments, known as *code clones*, are accumulated over time [10].

Cloned fragments in process models raise several issues. Firstly, clones make individual process models larger than they need to be, thus affecting their comprehensibility. Secondly, clones are modified independently, sometimes by different stakeholders,

leading to unwanted inconsistencies across models that originally contained a duplicate clone. Finally, process model clones hide potential efficiency gains. Indeed, by factoring out cloned fragments into separate subprocesses, and exposing these subprocesses as shared services, companies may reap the benefits of larger resource pools.

In this setting, this paper addresses the problem of retrieving all clones in a process model repository that can be refactored into shared subprocesses. Specifically, the contribution of the paper is an index structure, namely the *RPSDAG*, that provides operations for inserting and deleting models, as well as an operation for retrieving all clones in a repository that meet the following requirements:

- All retrieved clones must be single-entry, single-exit (SESE) fragments, since subprocesses are invoked according to a call-and-return semantics.
- All retrieved clones must be *exact* clones so that every occurrence can be replaced by an invocation to a single (shared) subprocess. While identifying *approximate* clones could be useful in some scenarios, approximate clones cannot be directly refactored into shared subprocesses, and thus fall outside the scope of this study.
- Any retrieved clone that occurs N times in the repository, should not be contained inside any other clone that also occurs N times. This maximality requirement is natural, since once we have identified a clone, every SESE fragment strictly contained inside this clone is also a clone, but we do not wish to return all such sub-clones.
- Retrieved clones must have at least two nodes (no "trivial" clones).

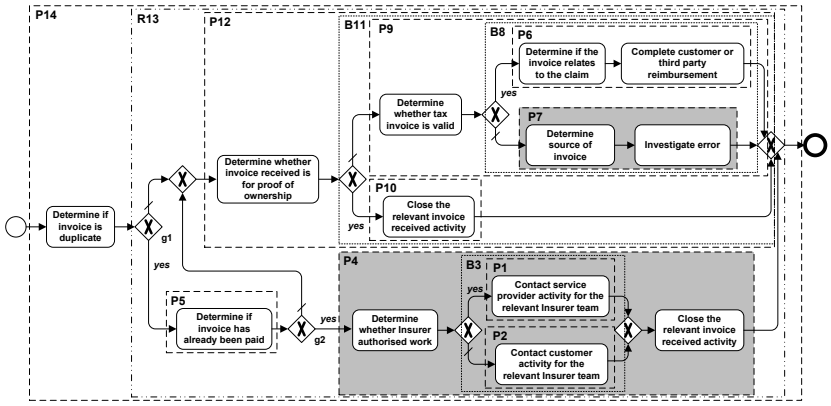
Identifying clones in a process model repository boils down to identifying fragments of a process model that are isomorphic to other fragments in the same or in another model. Hence, we need a method for decomposing a process model into fragments and a method for testing isomorphism of these fragments. Accordingly, the RPSDAG is built on top of two pillars: (i) a method for decomposing a process model into SESE fragments, namely the *Refined Process Structure Tree (RPST)* decomposition; and (ii) a method for calculating canonical codes for labeled graphs. These canonical codes reduce the problem of testing for graph isomorphism between a pair of graphs, to a string equality check. Section 2 introduces these methods and discusses how they are used to address the problem at hand. Next, Section 3 describes the RPSDAG, including its insertion and deletion algorithms. Section 4 presents an experimental evaluation of the RPSDAG using several process model repositories. Finally, Section 5 discusses related work while Section 6 draws conclusions.

2 Background

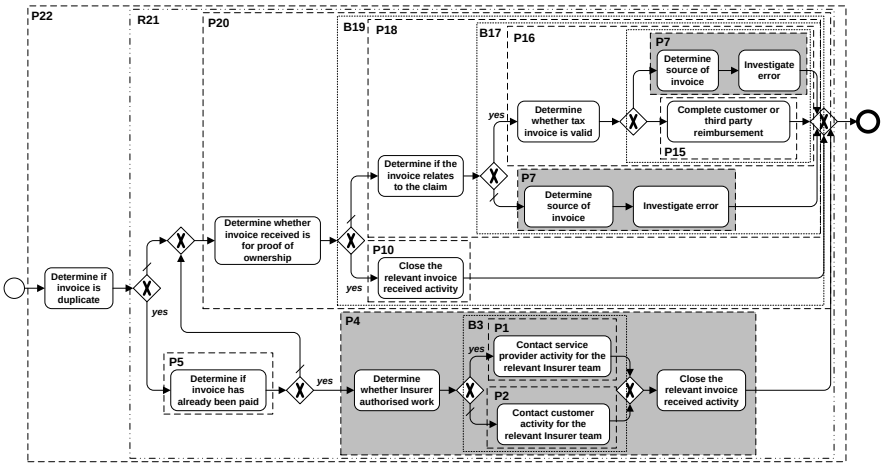
This section introduces the two basic ingredients of the proposed technique: the Refined Process Structure Tree (RPST) and the code-based graph indexing.

2.1 RPST

The RPST [18] is a parsing technique that takes as input a process model and computes a tree representing a hierarchy of SESE fragments. Each fragment corresponds to the subgraph induced by a set of edges. A SESE fragment in the tree contains all fragments at the lower level, but fragments at the same level are disjoint. As the partition is made in terms of edges, a single vertex may be shared by several fragments.



(a)



(b)

Fig. 1. Excerpt of two process models of an insurance company.

SESE fragments contained in the RPST can be classified into one of four classes [13]. A *trivial* (T) fragment consists of a single edge. A *polygon* (P) fragment is a sequence of fragments. A *bond* corresponds to a fragment where all child fragments share a common pair of vertices. Any other fragment is a *rigid*.

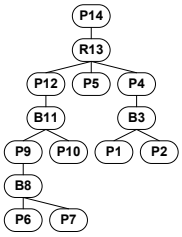


Fig. 2. RPST of process model in Fig. 1(a).

Figure 1(a) presents a sample process model, for which the RPST decomposition is shown in the form of dashed boxes. A simplified view of the tree is presented in Figure 2. Using this view, it can be seen that the root fragment corresponds to polygon P14, which in turn contains the rigid R13 and a set of trivial fragments (simple edges) which are not shown to simplify the figure. R13 is composed of polygons P12, P5, P4, and so forth. Polygons P4, P5, P7 and P10 have

been highlighted to ease the comparison with the similar subgraphs found in the process model shown in Figure 1(b).

Although the example process models are presented using the BPMN notation, the set of techniques can be used with other graph-oriented modeling notations. To achieve this notation-independence, we use the following graph-based representation of (BPMN) process models:

Definition 1 (Process Graph). A *Process Graph* is a labelled connected graph $G = (V, E, l)$, where:

- V is the set of vertices.
- $E \subseteq V \times V$ is the set of directed edges (e.g. representing control-flow relations).
- $l : V \rightarrow \Sigma^*$ is a labeling function that maps each vertex to a string over alphabet Σ. We distinguish the following special labels: $l(v) =$ "start" and $l(v) =$ "end" are reserved for start events and end events respectively; $l(v) =$ "xor-split" is used for vertices representing xor-split (exclusive decision) gateways; similarly $l(v) =$ "xor-join", $l(v) =$ "and-split" and $l(v) =$ "and-join" represent merge gateways, parallel split gateways and parallel join gateways respectively. For a task node t, $l(t)$ is the label of the task.

This definition can be extended to capture other types of BPMN elements by introducing additional types of nodes (e.g. a type of node for inclusive gateways). Organizational information such as lanes and pools can be captured by attaching dedicated attributes to each node (e.g. each node could have an attribute indicating the pool and lane to which it belongs). In this paper, we do not consider sub-processes, since each sub-process can be indexed separately for the purpose of clone identification.

2.2 Canonical labeling of graphs

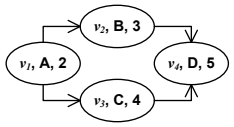
Our approach for graph indexing is an adaptation of the approach proposed in [20]. The adaptations we make relate to two specificities of process models that differentiate them from the class of graphs considered in [20]: (i) process models are directed graphs; (ii) process models can be decomposed into an RPST.

Following the method in [20], our starting point is a matrix representation of a process graph encoding the vertex adjacency and the vertex labels, as defined below.

Definition 2 (Augmented Adjacency Matrix of a Process Graph). Let $G = (V, E, l)$ be a Process Graph, and $\boldsymbol{v} = (v_1, \ldots, v_{|V|})$ a total order over the elements of V. The adjacency matrix of G, denoted as $\boldsymbol{A}$, is a $(0, 1)$-matrix such that $A_{i,j} = 1$ if and only if $(v_i, v_j) \in E$, where $i, j \in \{1 \ldots |V|\}$. Moreover, let us consider a function $h : \Sigma^* \rightarrow \mathbb{N} \setminus \{0, 1\}$ that maps each vertex label to a unique natural number greater than 1. The *Augmented Adjacency Matrix* $\boldsymbol{M}$ of G is defined as: $\boldsymbol{M} = \text{diag}(\ h(l(v_1)),\ \ldots,\ h(l(v_{|V|}))\ ) + \boldsymbol{A}$

Given the augmented adjacency matrix of a process graph (or a SESE fragment therein), we can compute a string (hereby called a *code*) by concatenating the contents of each cell in the matrix from left to right and from top to bottom. For illustration, consider graph G in Figure 3(a), which is an abstract view of fragment B3 of the running

example (cf. Figure 1(a)). For convenience, next to each vertex we show the unique vertex identifier (e.g. v_1), the corresponding label (e.g. $l(v_1)$ ="A"), and the numeric value associated with the label (e.g. $h(l(v_1)) = 2$). Assuming the order $\boldsymbol{v} = (v_1, v_2, v_3, v_4)$ over the set of vertices, the matrix shown in Figure 3(b) is the adjacency matrix of G. Figure 3(c) is the diagonal matrix built from $h(l(\boldsymbol{v}))$ whereas Figure 3(d) shows the augmented adjacency matrix $\boldsymbol{M}$ for graph G. It is now clear why 0 and 1 are not part of the codomain of function h, i.e. to avoid clashes with the representation of vertex adjacency. Figure 3(e) shows a possible permutation of $\boldsymbol{M}$ when considering the alternative order $\boldsymbol{v'} = (v_1, v_4, v_2, v_3)$ over the set of vertices.



(a)

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix} \quad \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 3 & 0 & 1 \\ 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 5 \end{pmatrix} \quad \begin{pmatrix} 2 & 0 & 1 & 1 \\ 0 & 5 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 1 & 0 & 4 \end{pmatrix}$$

(b) (c) (d) (e)

Fig. 3. (a) a sample graph, (b) its adjacency matrix, (c) its diagonal matrix with the vertex label codes, (d) its augmented adjacency matrix, and (e) a permutation of the augmented matrix

Next, we transform the augmented adjacency matrix into a code by scanning from left-to-right and top-to-bottom. For instance, the matrix in Figure 3(d) can be represented as "2.1.1.0.0.3.0.1.0.0.4.1.0.0.0.5". This code however does not uniquely represent the graph. If we chose a different ordering of the vertices, we would obtain a different code. To obtain a unique code (called a *canonical code*), we need to pick the code that lexicographically "precedes" all other codes that can be constructed from an augmented adjacency matrix representation of the graph. Conceptually, this means we have to consider all possible permutations of the set of vertices and compute a code for each permutation, as captured in the following definition.

Definition 3 (Graph Canonical Code). Let G be a process graph, $\boldsymbol{M}$ the augmented adjacency matrix of G. The *Graph Canonical Code* is the smallest lexicographical string representation of any possible permutation of matrix $\boldsymbol{M}$, that is:

$$\text{code}(\boldsymbol{M}) = \text{str}(\boldsymbol{P^T M P}) \big| \boldsymbol{P} \in \boldsymbol{\Pi} \wedge \forall \boldsymbol{Q} \in \boldsymbol{\Pi}, \boldsymbol{P} \neq \boldsymbol{Q} : \text{str}(\boldsymbol{P^T M P}) \prec \text{str}(\boldsymbol{Q^T M Q})$$

where:

- $\boldsymbol{\Pi}$ is the set of all possible permutations of the identity matrix $\boldsymbol{I}_{|M|}$
- str($\boldsymbol{N}$) is a function that maps a matrix $\boldsymbol{N}$ into a string representation.

Consider the matrices in Figures 3(d) and 3(e). The code of the matrix in Figure 3(e) is "2.0.1.1.0.5.0.0.0.1.3.0.0.1.0.4'. This code is lexicographically smaller than the code of the matrix in Figure 3(d) ("2.1.1.0.0.3.0.1.0.0.4.1.0.0.0.5"). If we explored all vertex permutations and constructed the corresponding matrices, we would find that "2.0.1.1.0.5.0.0.0.1.3.0.0.1.0.4" is the canonical code of the graph in Figure 3(a).

Enumerating all vertex permutations is unscalable (factorial on the number of vertices). Fortunately, optimizations can be applied by leveraging the characteristics of the graphs at hand. Firstly, by exploiting the nature of the fragments in an RPST, we can apply the following optimizations:

- The code of a polygon is computed in linear time by concatenating the codes of its contained vertices in the (total) order implied by the control flow.
- The code of a bond is also computed in linear time by taking the entry gateway as the first vertex, the exit gateway as the last vertex and all vertices in-between are ordered lexicographically based on their labels.

In the case of a rigid, we start by partitioning its vertices into two subsets: vertices with distinct labels, and vertices with duplicate labels. Vertices with distinct labels are deterministically ordered in lexicographic order. Hence we do not need to explore any permutations between these vertices. Instead, we can focus on deterministically ordering vertices with duplicate labels. Duplicate labels in process models arise in two cases: (i) there are multiple tasks (or events) with the same label; (ii) there are multiple gateways of the same type (e.g. multiple "xor-splits") that cannot be distinguished from one another since gateways generally do not have labels. To distinguish between multiple gateways, we pre-process each gateway g by computing the tasks that immediately precede it and the tasks that immediately follow it within the same rigid fragment, and in doing so, we skip other gateways found between gateway g and each of its preceding (or succeeding) tasks. We then concatenate the labels of the preceding tasks (in lexicographic order) and the labels of the succeeding tasks (again in lexicographic order) to derive a new label s_g. The s_g labels derived in this way are used to order multiple gateways of the same type within the same rigid. Consider for example g_1 and g_2 in Figure 1(b) and let s1 = "Determine if invoice has already been paid", s2="Determine whether invoice received is for proof of ownership" and s3="Determine whether Insurer authorized work". We have that s_{g_1} = "s1.s2" while s_{g_2} = "s1.s3.s2". Since s3 precedes s2, gateway g_2 will always be placed before g_1 when constructing an augmented adjacency matrix for R13. In other words, we do not need to explore any permutation where g_1 comes before g_2. Even if task "Determine if invoice is duplicate" precedes g_1 this is not used to compute s_{g_1} because this task is outside rigid R13. To ensure unicity, vertices located outside a rigid should not be used to compute its canonical code.

A similar approach is used to order tasks with duplicate labels within a rigid. This "label propagation" approach allows us to considerably reduce the number of permutations we need to explore. Indeed, we only need to explore permutations between multiple vertices if they have identical labels and they are preceded and followed by vertices that also have the same labels. The worst-case complexity for computing the canonical code is still factorial, but on the size of the largest group of vertices inside a rigid that have identical labels and identical predecessors and successors' labels.

3 Clone Detection Method

In this section we introduce the RPSDAG index structure and its associated clone retrieval, insertion and deletion methods.

3.1 Index structure and clone detection

The RPSDAG is designed to be directly implemented on top of standard relational databases. Accordingly, the RPSDAG consists of three tables: Codes(Code, Id, Size, Type), Roots(GraphId, RootId) and RPSDAG(ParentId, ChildId). Table Codes contains

the canonical code for each RPST fragment of an indexed graph. Column "Id" assigns a unique identifier to each indexed fragment, column Code gives the canonical code of a fragment, column Size is the number of vertices in the fragment, and column Type denotes the type of fragment ("p" for polygon, "r" for rigid and "b" for bond). The "Id" is auto-generated by incrementing a counter every time that a new code is inserted into the Codes table. Strictly speaking, the "Id" is redundant given that the code uniquely identifies each fragment. However, the "id" gives us a shorter way of uniquely identifying a fragment. When the canonical code of a new RPST node is constructed, we use the short identifiers of its child fragments as labels (in the diagonal of the augmented adjacency matrix), instead of using the child fragment's canonical codes which are longer.

Table Roots contains the Id of each indexed graph and the Id of the root fragment for that graph. Table RPSDAG is the main component of the index and is a combined representation of the RPSTs of all the graphs in the repository. Since multiple RPSTs may share fragments (these are precisely the clones we look for), the RPSDAG table represents a Directed Acyclic Graph (DAG) rather than a tree. Each tuple of this table is a pair consisting of a parent fragment id and a child fragment id. Figure 4 shows an extract of the tables representing the two graphs in Figure 1. Here, the fragment Ids have been derived from the fragment names in the Figure (e.g. the id for P4 is 4) and the codes have been hidden for the sake of readability.

Code	Id	Size	Type
-	1	1	p
-	2	1	p
-	3	4	b
-	4	6	p
-	5	1	p
-	6	2	p
...	...	...	...
-	13	20	r
-	14	21	p
...	...	...	...
-	22	24	p
...	...	...	...

ParentId	ChildId
3	1
3	2
4	3
8	6
8	7
...	...
13	4
13	5
...	...
14	13
16	7
16	15
17	7
21	4
21	5
...	...

GraphId	RootId
1	14
2	22

Fig. 4. RPST of process model in Fig. 1(a).

Looking at the RPSDAG, we can immediately observe that the maximal clones are those child fragments that have more than one parent. Accordingly, we can retrieve all clones in an RPSDAG by means of the following SQL query.

```
SELECT RPSDAG.ChildId, Codes.Size, COUNT(RPSDAG.ParentId)
  FROM RPSDAG, Codes
WHERE RPSDAG.ChildId = Codes.Id AND Codes.Size >= 2
GROUP BY RPSDAG.ChildId, Codes.Size
HAVING COUNT(RPSDAG.ParentId) >= 2;
```

This query retrieves the id, size and number of occurrences of fragments that have at least two parent fragments. Note that if a fragment appears multiple times in the indexed graphs, but always under the same parent fragment, it is not a maximal clone. For example, fragment P2 in Figure 1 always appears under B3, and B3 always appears under P4. Thus, neither P2 nor B3 are maximal clones, and the above query does not retrieve them. On the other hand, the query identifies P4 as a maximal clone since it has two parents (cf. tuples (13,4) and (21,4) in table RPSDAG in Fig. 4). Also, as per the requirements spelled out in Section 1, the query returns only fragments with at least 2 vertices. For example, even if there are two tuples with ChildId 5 and 10 in the example RPSDAG, the query will not return clones P5 and P10 as they are single nodes.

3.2 Insertion and Deletion

Algorithm 1 describes the procedure for inserting a new process graph into an indexed repository. Given an input graph, the algorithm first computes its RPST with function ComputeRPST() which returns the RPST's root node. Next, procedure InsertFragment is invoked on the root node to update tables Codes and RPSDAG. This returns the id of the root fragment. Finally, a tuple is added to table Roots with the id of the inserted graph and that of its root node.

Algorithm 1: Insert Graph

procedure InsertGraph(Graph m)
  RPST $root \Leftarrow$ ComputeRPST(m)
  $rid \Leftarrow$ InsertFragment($root$)
  $Roots \Leftarrow Roots \cup \{(\text{NewId}(), rid)\}$ // NewId() generates a fresh id

Algorithm 2: Insert Fragment

procedure InsertFragment(RPST f) returns RPSDAGNodeId
  {RPST, RPSDAGNodeId} $C \Leftarrow \varnothing$
  foreach RPST *child in* GetChildren(f) **do**
    $C \Leftarrow C \cup \{(child, \text{InsertFragment}(child))\}$
  $code \Leftarrow$ ComputeCode(f, C)
  $(id, type) =$ InsertNode($code, f$)
  foreach (cf, cid) *in* C **do**
    $RPSDAG \Leftarrow RPSDAG \cup \{(id, cid)\}$
  if $type =$*"p"* **then** InsertSubPolygons($id, code, f$)
  return id

Procedure InsertFragment (Algorithm 2) inserts an RPST fragment in table RPSDAG . This algorithm performs a depth-first search traversal of the RPST. Nodes are visited in postorder, i.e. a node is visited after all its children have been visited. When a node is visited, its canonical code is computed – function ComputeCode – based on the topology of the RPST fragment and the codes of its children (except of course for leaf nodes whose labels are their canonical codes). Next, procedure InsertNode is invoked to insert the node in table Codes, returning its id and type. This procedure (Algorithm 3) first checks if the node already exists in Codes, via function GetIdSizeType(). If it exists, GetIdSizeType() returns the id, size and type associated with that code, otherwise it returns the tuple (0,0,""). In this latter case, a fresh id is created for the node at hand, then its size and type are computed via functions ComputeSize and ComputeType, and a new tuple is added in table Codes. Function ComputeSize returns the sum of the sizes of all child nodes or 1 if the current node is a leaf. Once the node id and type have been retrieved, procedure InsertFragment adds a new tuple in table RPSDAG for each children of the visited node.

If the node is a polygon, procedure InsertSubPolygons() is invoked in the last step of InsertFragment. This procedure is used to identify common subpolygons and factor

Algorithm 3: Insert Node

```
procedure InsertNode(String code, RPST f) returns (RPSDAGNodeId, String)
    (id, size, type) ⇐ GetIdSizeType(code)
    if (id, size, type) = (0, 0, "") then
        id ⇐ NewId()
        size ⇐ ComputeSize(code)
        type ⇐ ComputeType(f)
        Codes ⇐ Codes ∪ {(code, id, size, type)}
    return (id,type)
```

Algorithm 4: Insert SubPolygons

```
procedure InsertSubPolygons(RPSDAGNodeId id, String code, RPST f)
    foreach (zcode, zid, zsize, ztype) in Codes such that ztype ="p" and zid ≠ id do
        {String} LCS ⇐ ComputeLCS(code, zcode)
        foreach lcode in LCS do
            (lid, ltype) = InsertNode(lcode, f)
            if lid ≠ id then RPSDAG ⇐ RPSDAG ∪ {(id, lid)}
            if lid ≠ zid then RPSDAG ⇐ RPSDAG ∪ {(zid, lid)}
            foreach sid in GetChildrenIds(lid) do
                RPSDAG ⇐ RPSDAG \ {(id, sid), (zid, sid)} ∪ {(lid, sid)}
            InsertSubPolygons(lid, lcode)
```

them out as separate nodes. Indeed, two polygons may share one or more subpolygons which should also be identified as clones. To illustrate this scenario, let us consider the two polygons P_1 and P_2 in Fig. 5, where $\text{code}(P_1) = B_2.a.B_1.w.z.a.B_1.c$ and $\text{code}(P_2) = a.B_1.c.d.a.B_1.w.z$.[1] These two polygons share bond B_1 as common child, while bond B_2 is a child of P_1 only. However, at a closer look, their canonical codes share three Longest Common Substrings (LCS), namely $a.B_1.c$, $a.B_1$ and $w.z$.[2] These common substrings represent common *subpolygons*, and thus clones that may be refactored as separate subprocesses.

Assume P_1 is already stored in the RPSDAG with children B_1 and B_2 (first graph in Fig. 5) and we now want to store P_2. We invoke procedure InsertFragment(P_2) and add a new node in the RPSDAG, with B_1 as a child (second graph in Fig. 5). Since P_2 is a polygon, we also need to invoke InsertSubPolygons(P_2). This procedure (Algorithm 4) retrieves all polygons from Codes that are different than P_2 (in our case there is only P_1). Then, for each such polygon, it computes all the non-trivial LCSs between its code and the code of the polygon just being inserted. This is performed by function ComputeLCS() which returns an ordered list of LCSs starting from the longest one (there are efficient algorithms to compute the LCSs of two strings. This can be done

[1] For the sake of readability, here we use the node/fragment label as canonical code, instead of the numeric value associated with their label

[2] An LCS of strings S_1 and S_2 is a substring of both S_1 and S_2, which is not contained in any other substring shared by S_1 and S_2. LCSs of size one are not considered for obvious reasons.

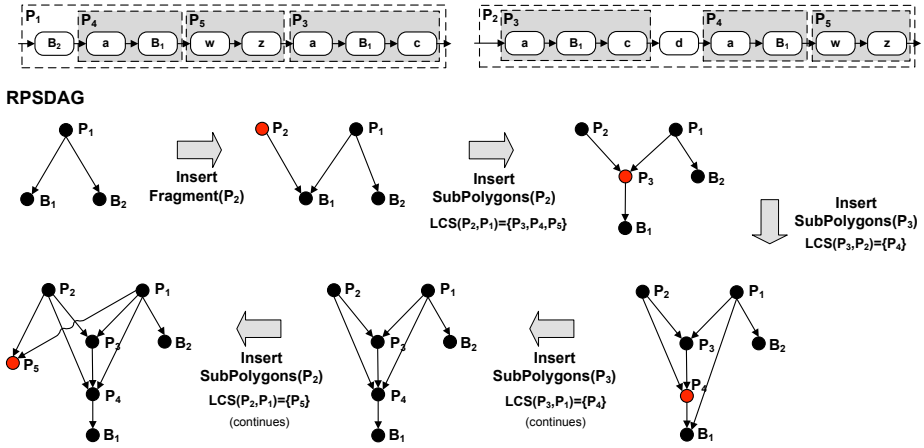


Fig. 5. Two subpolygons and the corresponding RPSDAG.

by using a *suffix tree* [17] to index the canonical codes of the existing polygons, see e.g. [6] for a survey of LCS algorithms). In the example at hand, the longest substring is $a.B_1.c$. This substring can be seen as the canonical code of a "shared" subpolygon between P_2 and P_1. To capture the fact that this shared subpolygon is a clone, we insert a new node P_3 in the RPSDAG with canonical code $a.B_1.c$ (unless such a node already exists, in which case we reuse the existing node) and we insert an edge from P_2 to P_3 and from P_1 to P_3. In order to avoid redundancy in the RPSDAG, the new node P_3 then "adopts" all the common children that it shares with P_2 and with P_1. These nodes are retrieved by function GetChildrenIds(), which simply returns the ids of all child nodes of P_3: these will either be children of P_1 or P_2 or both. In our example, there is only one common child, namely B_1, which is adopted by P_3 (see third graph in Fig. 5).

It is possible that the newly inserted node also shares subpolygons with other nodes in the RPSDAG. So, before continuing to process the remaining LCSs between P_1 and P_2, we invoke procedure InsertSubPolygons recursively over P_3. In our example, the code of P_3 shares the substring $a.B_1$ with both P_2 and P_1 (after excluding the code of P_3 from the codes of P_2 and P_1 since P_3 is already a child of these two nodes). Thus we create a new node P_4 with code $a.B_1$ as a child of P_3 and P_2, and we make P_4 adopt the common child B_1 (see fourth graph in Fig. 5). We repeat the same operation between P_3 and P_1 but since this time P_4 already exists in the RPSDAG, we simply remove the edge between P_1 and B_1, and add an edge between P_1 and P_4 (fifth graph in Fig. 5). Then, we resume the execution of InsertSubPolygons(P_2) and move to the second LCS between P_1 and P_2, i.e. $a.B_1$. Since this substring has already been inserted into the RPSDAG as node P_4, nothing is done. This process of searching for LCSs is repeated until no more non-trivial common substrings can be identified. In the example at hand, we also add subpolygon $w.z$ between P_1 and P_2 (last graph in Fig. 5). At this point, we have identified and factored out all maximal subpolygons shared by P_1 and P_2, and we can repeat the above process for other polygons in the RPSDAG whose canonical code shares a common substring with that of P_1.

Sometimes there may be multiple overlapping LCSs of the same size. For example, given the codes of two polygons $a.b.c.d.e.f.a.b.c$ and $a.b.c.k.b.c.d$, if we extract one substring (say $a.b.c$), we can no longer extract the second one ($b.c.d$). In these cases we locally choose based on the number of occurrences of an LCS within the two strings in question. If they have the same number of occurrences, we randomly choose one. In the example above, $a.b.c$ has the same size of $b.c.d$ but it occurs 3 times, so we pick $a.b.c$.

We do not explicitly discuss the cases where the polygon to be inserted is a subpolygon or superpolygon of an existing polygon. These are just special cases of shared subpolygon detection, and they are covered by procedure InsertSubPolygons.

Algorithm 5 shows the procedure for deleting a graph from an indexed repository. This procedure relies on another procedure for deleting a fragment, namely DeleteFragment (Algorithm 6). The DeleteFragment procedure performs a depth-first search traversal of the RPSDAG, visiting the nodes in post-order. Nodes with at most one parent are deleted, because they correspond to fragments that appear only in the deleted graph. Deleting a node entails deleting the corresponding tuple in table Codes and deleting all tuples in the RPSDAG table where the deleted node corresponds to the parent id. If a fragment has two or more parents, the traversal stops along that branch since this node and its descendants must remain in the RPSDAG. At the end of the DeleteGraph procedure, the graph itself is deleted from table Roots through its root id.

If the node to be removed is a polygon, it has only two parents and both parents are polygons, it is a shared subpolygon. Before deleting such a node, we need to make sure all its children are adopted by the parent polygon that will remain in the RPSDAG. For space reasons, we omit this step in the deletion algorithm.

Also, for space reasons the above algorithms do not take into account the case where a fragment is contained multiple times in the same parent fragment, like for example a polygon that contains two tasks with identical labels. To address this case, we have to introduce an attribute "Weight" in the RPSDAG table, representing the number of times a child node occurs inside a parent node. Under this representation, a node is a clone if the sum of the weights of its incoming edges in the RPSDAG is greater than one.

Complexity. The deletion algorithm performs a depth-first search, which is linear on the size of the graph being deleted. Similarly, the insertion algorithm traverses the inserted graph in linear time. Then, for each fragment to be inserted, it computes its code. Computing the code is linear on the size of the fragment for bonds and polygons, while for rigids it is factorial on the largest number of vertices inside the rigid that share identical labels, as discussed in Section 2.2.[3] Finally, if the fragment is a polygon, we compute all LCSs between this polygon and all other polygons in the RPSDAG. Using a suffix tree, this operation is linear on the sum of the lengths of all polygons' canonical codes.

4 Evaluation

We evaluated the RPSDAG using four datasets: the collection of SAP R3 reference process models [9], a model repository obtained from an insurance company under condition of anonymity and two collections from the IBM BIT process library [5], namely collections A and B3. In the BIT process library there are 5 collections (A, B1, B2, B3

[3] A tighter complexity bound for this problem is given in [1].

Algorithm 5: Delete Graph

```
procedure DeleteGraph(GraphId mid)
    RPSDAGNodeId rid ⇐ GetRoot(mid)
    DeleteFragment(rid)
    Roots ⇐ Roots \ {(mid, rid)}
```

Algorithm 6: Delete Fragment

```
procedure DeleteFragment(RPSDAGNodeId fid)
    if |{(pid, cid) ∈ RPSDAG : cid = fid}| ≤ 1 then
        foreach (pid, cid) in RPSDAG where pid = fid do
            DeleteFragment(cid)
            RPSDAG ⇐ RPSDAG \ {(pid, cid)}
        Codes ⇐ {(code, id, size, type) ∈ Codes : id ≠ fid}
```

and C). We excluded collections B1 and B2 because they are earlier versions of B3, and collection C because it is a mix of models from different sources and as such it does not contain any clones. The SAP repository contains 595 models with sizes ranging from 5 to 119 nodes (average 22.28, median 17). The insurance repository contains 363 models ranging from 4 to 461 nodes (average 27.12, median 19). The BIT collection A contains 269 models ranging from 5 to 47 nodes (average 17.01, median 16) while collection B3 contains 247 models with 5 to 42 nodes (average 12.94, median 11).

Performance evaluation. We first evaluated the insertion times. Obviously inserting a new model into a nearly-empty RPSDAG is less costly than doing so in an already populated one. To factor out this effect, we randomly split each dataset as follows: One third of the models were used to construct an initial RPSDAG and the other two-thirds were used to measure insertion times. In the SAP repository, 200 models were used to construct an initial RPSDAG. Constructing the initial RPSDAG took 26.1s. In the insurance company repository, the initial RPSDAG contained 121 models and its construction took 7.4s. For the BIT collections A and B3, 90 and 82 models respectively were used for constructing the initial RPSDAG. Constructing the initial RPSDAGs took 4.2s for collection A and 3.5s for collection B3. All tests were conducted on a PC with a dual core Intel processor, 1.8 GHz, 4 GB memory, running Microsoft Windows 7 and Oracle Java Virtual Machine v1.6. The RPSDAG was implemented as a Java console application on top of MySQL 5.1. Each test was run 10 times and the execution times obtained across the ten runs were averaged.

Table 1 summarizes the insertion time per model for each collection (min, max, avg., std. dev., and 90^{th} percentile). All values are in milliseconds. These statistics are given for two cases: without subpolygon clone detection and with subpolygon clone detection. We observe that the average insertion time per model is about 3-5 times larger when subpolygon clone detection is performed. This overhead comes from the step where we compare inserted polygon with each already-indexed polygon and compute the longest-common substring of their canonical codes. Still, the average execution

times remain in the order of tens of milliseconds across all model collections even with subpolygon detection. The highest average insertion time (125ms) is observed for the Insurance collection. This collection contains some models with large rigid components in their RPST. In particular, one model contained a rigid component in which 2 task labels appeared 9 times each – i.e. 9 tasks had one of these labels and 9 tasks had the other – and these tasks were all preceded by the same gateway. This label repetition affected the computation of the canonical code (4.4 seconds for this particular rigid). Putting aside this extreme case, all insertion times were under one second and in 90% of the cases (l_{90}), the insertion times in this collection were under 50ms without subpolygon detection and 222ms with subpolygon detection. Thus we can conclude that the proposed technique scales up to real-sized model collections.

	min	max	avg	std	l_{90}	
SAP	4	85	20	14	40	No subpolygon
Insurance	5	1722	32	113	49	
BIT A	3	58	12	9	23	
BIT B3	5	467	14	9	28	
SAP	4	482	97	81	202	Subpolygon
Insurance	26	4402	126	291	222	
BIT A	18	128	41	20	59	
BIT B3	5	150	33	24	69	

Table 1. Model insertion times (in ms).

As explained in Section 3, once the models are inserted, we can find all clones with a SQL query. The query to find all clones with at least 2 nodes and 2 parents from the SAP reference models takes 75ms on average (90ms for the insurance models, 118 and 116ms for BIT collections A and B3).

Refactoring gain. One of the main applications of clone detection is to refactor the identified clones as shared subprocesses in order to reduce the size of the model collection and increase its navigability. In order to assess the benefit of refactoring clones into subprocesses, we define the following measure.

Definition 4. *The* refactoring gain *of a clone is the reduction in number of nodes obtained by encapsulating that clone into a separate subprocess, and replacing every occurrence of the clone with a task that invokes this subprocess. Specifically, let S be the size of a clone, and N the number of occurrences of this clone. Since all occurrences of a clone are replaced by a single occurrence plus N subprocess invocations, the refactoring gain is: $S \cdot N - S - N$.*

Given a collection of models, the total refactoring gain *is the sum of the refactoring gains of the clones of non-trivial clones (size ≥ 2) in the collection.*

Table 2 summarizes the total refactoring gain for each model collection. The first two columns correspond to the total number of clones detected and the total refactoring gain without subpolygon refactoring. The third and fourth column show number of clones and refactoring gain with subpolygon refactoring. The table shows that a significant number of clones can be found in all model collections, and that the size of these model collections could be reduced by 4-14% if clones were factored out into shared subprocesses. The table also shows that subpolygon clone detection adds significant value to the clone detection method. For instance, in the case of the insurance

models, we obtain about 3 times more clones and 3 times more refactoring gain when subpolygon clone detection is performed.

	No subpolygon		With subpolygon	
	Nr. clones	refactoring gain	Nr. clones	refactoring gain
SAP	204	1359 (10.3%)	479	1834 (13.8%)
Insurance	107	394 (4%)	279	883 (9%)
BIT A	57	195 (4.3%)	174	384 (8.4%)
BIT B3	19	208 (6.5%)	49	259 (6.6%)

Table 2. Total refactoring gain without and with subpolygon refactoring.

Table 3 shows more detailed statistics of the clones found with subpolygon detection. The first three columns give statistics about the sizes of the clones found, the next three columns refer to the frequency (number of occurrences) of the clones, and the last three correspond to refactoring gain. We observe that while the average clone size is relatively small (3-5 nodes), there are some large clones with sizes of 30+ nodes.

	Size			# occurrences			Refactoring gain		
	avg	max	std. dev.	avg	max	std. dev.	avg	max	std. dev.
SAP	4.79	41	4.05	2.33	8	0.77	3.83	44	5.68
Insurance	3.58	32	3.18	2.76	41	3.18	3.16	79	7.67
BIT A	3.02	16	2.08	2.75	9	1.29	2.21	15	3.12
BIT B3	3.18	9	1.58	3.82	20	3.70	5.29	37	8.87

Table 3. Statistics of detected clones (with subpolygon detection).

5 Related Work

Clone detection in software repositories has been an active field for several years. According to [3], approaches can be classified into: textual comparison, token comparison, metric comparison, abstract syntax tree (AST) comparison, and program dependence graphs (PDG) comparison. The latter two categories are close to our problem, as they use a graph-based representation. In [2], the authors describe a method for clone detection based on ASTs. The method applies a hash function to subtrees of the AST in order to distribute subtrees across buckets. Subtrees in the same bucket are compared by testing for tree isomorphism. This work differs from ours in that RPSTs are not perfect trees. Instead, RPSTs contain rigid components that are irreducible and need to be treated as subgraphs—thus tree isomorphism is not directly applicable. [11] describes a technique for code clone detection using PDGs. A subgraph isomorphism algorithm is used for clone detection. In contrast, we employ canonical codes instead of pairwise subgraph isomorphism detection. Another difference is that we take advantage of the RPST in order to decompose the process graph into SESE fragments.

Work on clone detection has also been undertaken in the field of model-driven engineering. [4] describes a method for detecting clones in large repositories of Simulink/-TargetLink models from the automotive industry. Models are partitioned into connected components and compared pairwise using a heuristic subgraph matching algorithm. Again, the main difference with our work is that we use canonical codes instead of subgraph isomorphism detection. In [12], the authors describe two methods for exact and approximate matching of clones for Simulink models. In the first method, they apply an

incremental, heuristic subgraph matching algorithm. In the second approach, graphs are represented by a set of vectors built from graph features: e.g. path lengths, vertex in/out degrees, etc. An empirical study shows that this feature-based approximate matching approach improves pre-processing and running times, while keeping a high precision. However, this data structure does not support incremental insertions/deletions.

Our work is also related to graph database indexing. GraphGrep [16] is an index designed to retrieve paths in a graph that match a given regular expression. However, paths are indexed up to a certain threshold length, reducing the usefulness of this index for clone detection. The closure-tree index [7] organizes graphs as a hierarchy of subgraphs. Each leaf in the tree stores (a reference to) an entire graph. Non-leave nodes contain closure subgraphs, where nodes are either real nodes in the original graph or "closure" nodes representing folded subgraphs. The major drawbacks of the closure-tree index are large store requirements, and a significant time overhead during creation/update.

In [8], we described an index to retrieve process models in a repository that exactly or approximately match a given model fragment. In this approach, paths in the process models are used as index features. Given a collection of models, a B+ tree is used to reduce the search space by discarding those models that do not contain any path of the query model. The remaining models are checked for subgraph isomorphism.

In [19], eleven process model refactoring techniques are identified and evaluated. Extracting process fragments as subprocesses is one of the techniques identified. Our work addresses the problem of identifying opportunities for such "fragment extraction".

6 Conclusion

We presented a technique to index process models in order to identify duplicate SESE fragments (clones) that can be refactored into shared subprocesses. The proposed index, namely the RPSDAG, combines a method for decomposing process models into SESE fragments (the RPST decomposition) with a method for generating a unique string from a labeled graph (canonical codes). These canonical codes are used to determine whether a SESE fragment in a model appears elsewhere in the same or in another model.

The RPSDAG has been implemented and tested using process model repositories from industrial practice. In addition to demonstrating the scalability of the RPSDAG, the experimental results show that a significant number of non-trivial clones can be found in industrial process model repositories. In one repository, ca. 480 non-trivial clones were found. By refactoring these clones, the overall size of the repository is reduced by close to 14%, which arguably would enhance the repository's maintainability.

A standalone release of the RPSDAG implementation, together with sample models, is available at: `http://apromore.org/tools`. The tool takes as input a collection of files and produces a listing of all clones found.

The current RPSDAG implementation can be further optimized in three ways: (i) by using suffix trees for identifying the longest common substrings between the code of an inserted polygon and those of already-indexed polygons; (ii) by storing the canonical codes of each fragment in a hash index in order to speed up retrieval; (iii) by using the Nauty library for computing canonical codes[4]. Nauty implements several optimizations that potentially complement the optimizations described in Section 2.2.

[4] `http://cs.anu.edu.au/~bdm/nauty/`

Another avenue for future work is to extend the proposed technique in order to identify *approximate* clones. This has applications in the context of process standardization, when analysts seek to identify similar but non-identical fragments and to replace them with standardized fragments in order to increase the homogeneity of work practices.

Acknowledgments This research is partly funded by the Estonian Science Foundation, the European Social Fund via the Estonian ICT Doctoral School and the European Regional Development Fund via the Estonian Centre of Excellence in Computer Science.

References

1. L. Babai. Monte carlo algorithms in graph isomorphism testing. Technical Report D.M.S. No. 79-10, Universite de Montreal, 1979.
2. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *the Int. Conf. on Software Maintenance*, pages 368–377, 1998.
3. S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. on Software Engineering*, 33(9):577–591, 2007.
4. F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone Detection in Automotive Model-based Development. In *ICSE*, 2008.
5. D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Instantaneous soundness checking of industrial business process models. In *BPM*, 2009.
6. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
7. H. He and A.K. Singh. Closure-Tree: An Index Structure for Graph Queries. In *the 22nd Int. Conf. on Data Engineering*. IEEE Computer Society, 2006.
8. T. Jin, J. Wang, N. Wu, M. La Rosa, and A.H.M. ter Hofstede. Efficient and Accurate Retrieval of Business Process Models through Indexing. In *OTM*, 2010.
9. G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley, 1998.
10. R. Koschk. Identifying and Removing Software Clones. In Tom Mens and Serge Demeyer, editors, *Software Evolution*. Springer, 2008.
11. J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *WCRE*, 2001.
12. N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and Accurate Clone Detection in Graph-based Models. In *the 31st Int. Conf. on Software Engineering*, pages 276–286. IEEE Computer Society, 2009.
13. A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified Computation and Generalization of the Refined Process Structure Tree. In *WSFM*, 2010.
14. H. A. Reijers, R. S. Mans, and R. A. van der Toorn. Improved Model Management with Aggregated Business Process Models. *Data Knowl. Eng.*, 68(2):221–243, 2009.
15. M. Rosemann. Potential pitfalls of process modeling: Part a. *Business Process Management Journal*, 12(2):249–254, 2006.
16. D. Shasha, J.T-L. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In *PODS*, pages 39–52, 2002.
17. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
18. J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. *Data Knowl. Eng.*, 68(9):793–818, 2009.
19. B. Weber and M. Reichert. Refactoring Process Models in Large Process Repositories. In *CAiSE*, pages 124–139. Springer, 2008.
20. D. W. Williams, J. Huan, and W. Wang. Graph Database Indexing Using Structured Graph Decomposition. In *ICDE*, 2007.